DISPENSA SVIGRUPPO

Git:

* chmod:
  + chmod +x [nomeScript] :: esegue lo script fornito (per ricostruire la repo)
* git add:
  + git add [fileName/directoryName] :: aggiunge all’index i file/directory indicate
* git branch:
  + git branch :: mostra tutti i branch presenti
  + git branch [branchName] :: crea un branch con nome branchName
  + git branch -d [branchName] :: elimina il branch indicato da branchName
  + git branch -m master main :: muove/rinomina il ramo
* git cat-file:
  + git cat-file -p [hashCommit/hashTree/hashFile] :: mostra il contenuto del commit/tree/blob di cui gli viene passato l’hash

NB se un commit non ha parent vuol dire che è stato il primo commit effettuato!

NB se un commit ha due parent il primo dall’alto indica il branch su cui è stato fatto il merge!

* + git cat-file -- --batch-check --batch-all-objects

Mostra tutti gli oggetti

* git cherry-pick:
  + git cherry-pick [hashCommit/tagName] :: prende il commit indicato e lo aggiunge come nuovo commit al branch corrente
* git commit:
  + git commit -m [message] :: effettua il commit con messaggio indicato

NB gli hash dei commit già presenti non vanno presi in considerazione dato che vengono calcolati tenendo conto di data e autore (per verificare correttezza repository vanno considerati solo hash dei tree ed hash dei blob)

* + git commit –amend -m [message] :: Sostituisco il messaggio del commit precedente e gli staged changes.
    - * Creo un nuovo commit uguale al precedente e cambio il messaggio di log e aggiungo gli staged changes.
      * Il commit precedente viene rimosso, si crea un dangling commit
* git fsck:
  + git fsck :: restituisce tutti i dangling blob e i dangling commit

NB per ottenere un dangling blob basta creare un file, aggiungerlo all’index e poi effettuare git reset, mentre nel caso di un dangling commit serve fare un commit e poi un reset a un commit precedente!

* git hash-object:
  + git hash-object [fileName] :: restituisce l’hash del file indicato
* git log:
  + git log :: mostra tutti i commit del branch corrente
  + git log [branchName] :: fa vedere i commit del branch indicato
  + git log --all --decorate --oneline --graph :: mostra sintesi repository come albero
  + git log --all --graph : mostra repository estesa come albero

NB se un commit non ha parent vuol dire che è stato il primo commit effettuato!

* git merge:
  + git merge [branchName] -m [commitMessage] :: effettua merge sul branch corrente tra ultimo commit del branch corrente e l’ultimo commit del branch indicato
  + git merge [tagName] -m [commitMessage] :: effettua merge sul branch corrente tra ultimo commit del branch corrent e il commit individuato dal tag sul nuovo commit
  + git merge --no-ff [branchName/tagName] -m [commitMessage] :: effettua merge sul branch corrente tra ultimo commit del branch corrente e l’ultimo commit del branch indicato o il commit individuato dal tag, dove --no-ff garantisce che venga creato un commit nuovo!

NB non è detto che una merge crei un nuovo commit, bensì questa proprietà deve essere esplicitato nei parametri!

* git rebase:
  + git rebase [branchName] :: effettua il rebase del branch corrente spostandone il primo commit in seguito al branch indicato
* git reset:

git reset [commit] :: ritorna al commit indicato e resetta solo l’index allo stato del commit indicato

git reset --soft [commit] :: ritorna al commit indicato ma non resetta nè working directory nè index

git reset --hard [commit] :: ritorna al commit indicato e resetta sia working directory sia index allo stato del commit indicato

* git stash:
  + git stash :: salva index e working directory in una pila LIFO
  + git stash pop :: ripristina index e working directory da pila LIFO
* git status:
  + git status :: mostra lo stato corrente di working directory e index
* git switch:
  + git switch [branchName] :: sposta la testa sul branch indicato
  + git switch -d [tagName] :: fai puntare la HEAD al commit con il tag indicato

NB per redigere lo script di creazione della repository se si deve spostare la HEAD su commit non puntati da branch bisogna usare i tag invece degli hash, dato che quest’ultimi vengono calcolati diversamente in base a ora e autore!

* git tag:
  + git tag [tagName] :: applica il tag tagName al commit più recente sul branch corrente
  + git tag -d [tagName] :: elimina il tag indicato

// Modifica dei file

* mkdir:
  + mkdir [folderName] :: crea una cartella vuota
* printf:
  + printf [string] > [fileName] :: scrive la stringa nel file indicato, sovrascrivendolo o creandolo se non esiste
  + pritnf [string] >> [fileName] :: appenda la stringa nel file indicato, creandolo se non esiste
* cp:
  + cp [fileToCopy] [listOfFilesToCopyIn] :: copia il file indicato nei file indicati in seguito, sovrascrivendoli o creandoli se non presenti
* mv
  + mv [file][direcotry]

Move or rename a file, directory

Eiffel:

* Parole riservate:
  + and :: clausola AND
  + ANY :: è la classe root di Eiffel, come la classe Object in Java
  + attached :: serve per controllare che l’oggetto riferito non sia void (ex: *if attached x as x\_is\_*attached *then x\_is\_attached [istruzioni] end*)
  + create :: clausola a cui segue la dichiarazione dei costruttori da implementare nelle features successive
  + Current :: referenzia l’oggetto corrente e coincide con il this di Java
  + deferred :: significa che la classe/metodo è astratta/o (come abstract in Java)
  + detachable :: significa che l’oggetto/variabile indicata in seguito può essere Void
  + end :: serve per concludere ogni dichiarazione di classe/override/feature/do/if-elseif-else/loop
  + ensure :: serve per definire le postcondizioni
  + ensure then :: mette in AND le postcondizioni della sopraclasse e le postcondizioni della classe corrente (le postcondizioni di una sottoclasse si possono solo rafforzare ma non indebolire: “require no more, promise no less”)
  + feature :: introduce la dichiarazione di attributi e metodi della classe
  + for loop :: ciclo for che in Eiffel è costuito dalle 3 parole *from, until* e *loop* (from [condizioni] until [condizioni] loop [istruzioni] end)
  + if/elseif/else :: istruzioni utilizzate per testare condizioni (if [condizioni] then [istruzioni] elseif [condizioni] then [istruzioni] else [istruzioni] end)
  + implies :: [condizione] implies [conseguenza] (non va if prima della condizione indicata e si usa di spesso nelle postcondizioni per determinare lo stato corrente dell’oggetto in base a quello che era lo stato iniziale)
  + inherit :: coincide con l’ “extends” di Java ed è seguito dalla dichiarazione delle classi ereditate
  + invariant :: clausola a seguito della quale vengono definiti gli invarianti di classe, ovvero le proprietà che devono essere valide in ogni stato dell’oggetto e che vengono ereditate da tutte le eventuali sottoclassi e aggiunte in AND alle ulteriori invarianti dichiarate da quest’ultime (gli invarianti di una sottoclasse si possono solo rafforzare ma non indebolire: “require no more, promise no less”)
  + local :: clausola sotto cui dichiarare variabili locali
  + not :: clausola NOT
  + old :: parola riservata utilizzata nelle postcondizioni per indicare uno stato o una proprietà dell’oggetto precedente alla chiamate della funzione
  + or :: clausola OR
  + redefine :: dichiara i metodi di cui si effettuerà l’override nelle features
  + require :: serve per definire le precondizioni
  + require else :: mette in OR le precondizioni della sopraclasse e le precondizioni della classe corrente (le precondizioni di una sottoclasse si possono solo indebolire ma non rafforzare: “require no more, promise no less”)
  + rescue :: coincide con il catch di Java ed è usato per gestire eccezioni
  + Result :: indica il valore di ritorno
  + retry :: se presente è l’ultima delle istruzioni correttive della rescue e permette di provare a rieseguire il metodo una volta gestita l’eccezione
  + Void :: indica riferimento nullo e coincide con il null in Java
* Simboli:
  + := :: operatore di assegnamento
  + = :: operatore di confronto
  + /= :: not equal